Merge vs. Rebase – Der fundamentale Unterschied

Der Unterschied zwischen `git merge` und `git rebase` ist eines der wichtigsten Konzepte, das du als fortgeschrittener Git-Nutzer wirklich verstanden haben solltest. Beide Befehle dienen demselben Zweck – **Änderungen aus einem Branch in einen anderen zu integrieren** – aber sie tun dies auf völlig unterschiedliche Weise, mit unterschiedlichen Auswirkungen auf deine Commit-Historie.

Die Ausgangssituation

Stellen wir uns folgendes Szenario vor: Du arbeitest an einem Feature-Branch, während dein Team weiterhin Commits auf `main` macht. Nach einiger Zeit sieht deine Repository-Struktur so aus:

```
gitGraph
    commit id: "A"
    commit id: "B"
    branch feature
    commit id: "X"
    commit id: "Y"
    checkout main
    commit id: "C"
    commit id: "D"
```

Du hast auf deinem `feature`-Branch die Commits **X** und **Y** erstellt, während auf `main` in der Zwischenzeit **C** und **D** hinzugekommen sind. Jetzt möchtest du die Änderungen von `main` in deinen Feature-Branch integrieren (oder umgekehrt deinen Feature-Branch in `main` einbringen). Hier kommen Merge und Rebase ins Spiel.

Was passiert bei einem Merge?

Ein **Merge** erstellt einen neuen *Merge-Commit*, der die beiden Entwicklungslinien zusammenführt. Dieser Merge-Commit hat **zwei Eltern-Commits** – er verbindet buchstäblich die beiden Branches miteinander.

Der Merge-Befehl

```
# Du bist auf feature und möchtest main integrieren
git checkout feature
git merge main
```

In PhpStorm: **Git → Merge... → main auswählen**

Das Ergebnis nach dem Merge

```
gitGraph
    commit id: "A"
    commit id: "B"
    branch feature
    commit id: "X"
    commit id: "Y"
    checkout main
    commit id: "C"
    commit id: "D"
    checkout feature
    merge main id: "M" tag: "Merge-Commit"
```

Der neue Commit **M** ist der Merge-Commit. Er enthält keine eigenen Code-Änderungen (außer eventuell Konfliktlösungen), sondern dient als „Knotenpunkt", der die beiden Historien verbindet.

Eigenschaften eines Merge

Aspekt	Beschreibung
Commit-Historie	Bleibt vollständig erhalten, zeigt aber eine verzweigte Struktur
Originalcommits	X und Y behalten ihre ursprünglichen SHA-Hashes
Neuer Commit	Ein Merge-Commit mit zwei Eltern wird erstellt
Nicht-destruktiv	Die bestehende Historie wird niemals verändert

Was passiert bei einem Rebase?

Ein **Rebase** nimmt deine Commits und „verpflanzt" sie auf eine neue Basis. Statt eines Merge-Commits werden deine Commits *neu erstellt* – sie werden auf den aktuellen Stand des Ziel-Branches aufgebaut, als hättest du erst jetzt mit der Arbeit begonnen.

Der Rebase-Befehl

```
# Du bist auf feature und möchtest auf main "umbasieren"
git checkout feature
git rebase main
```

In PhpStorm: **Git → Rebase... → main auswählen**

Was genau passiert beim Rebase?

Der Rebase-Prozess läuft in mehreren Schritten ab:

1. **Git identifiziert die Commits**, die nur auf `feature` existieren (X und Y)
2. **Git speichert diese Commits** temporär (als Patches)
3. **Git setzt den `feature`-Branch** auf die Spitze von `main` (Commit D)
4. **Git wendet die gespeicherten Commits** nacheinander wieder an

Das Ergebnis sind *neue Commits* **X'** und **Y'**, die denselben Inhalt haben wie die Originale, aber:

- einen **anderen Eltern-Commit** (sie bauen jetzt auf D auf, nicht auf B)
- einen **anderen SHA-Hash** (weil sich der Eltern-Commit geändert hat)
- einen **anderen Zeitstempel** (optional, je nach Konfiguration)

Das Ergebnis nach dem Rebase

```
gitGraph
    commit id: "A"
    commit id: "B"
    commit id: "C"
    commit id: "D"
    commit id: "X'" tag: "neu erstellt"
```

```
    commit id: "Y'" tag: "neu erstellt"
```

Beachte: Die ursprünglichen Commits X und Y existieren technisch noch im Repository (erreichbar über `git reflog`), aber kein Branch zeigt mehr auf sie. Sie werden bei der nächsten Garbage Collection entfernt.

Visueller Vergleich: Merge vs. Rebase

Um den Unterschied noch deutlicher zu machen, hier beide Ergebnisse im direkten Vergleich:

```
flowchart TB
    subgraph vorher["Ausgangssituation"]
        direction LR
        A1["A"] --> B1["B"]
        B1 --> C1["C"] --> D1["D"]
        B1 --> X1["X"] --> Y1["Y"]
    end

    subgraph merge["Nach git merge main"]
        direction LR
        A2["A"] --> B2["B"]
        B2 --> C2["C"] --> D2["D"]
        B2 --> X2["X"] --> Y2["Y"]
        D2 --> M["M"]
        Y2 --> M
    end

    subgraph rebase["Nach git rebase main"]
        direction LR
        A3["A"] --> B3["B"] --> C3["C"] --> D3["D"]
        D3 --> X3["X'"] --> Y3["Y'"]
    end

    vorher --> merge
    vorher --> rebase
```

```
    style M fill:#90EE90,color:#000000
    style X3 fill:#FFD700,color:#000000
    style Y3 fill:#FFD700,color:#000000
```

Die tiefgreifenden Unterschiede im Detail

1. Integrität der Historie vs. Sauberkeit

Merge bewahrt die *exakte historische Wahrheit*: Es zeigt, dass parallel entwickelt wurde und wann die Zusammenführung stattfand. Das kann wertvoll sein, wenn du später nachvollziehen möchtest, wie die Entwicklung tatsächlich ablief.

Rebase erzeugt eine *saubere, lineare Historie*, als hätte es nie parallele Entwicklung gegeben. Das macht die Historie leichter lesbar, „lügt" aber technisch gesehen über den tatsächlichen Entwicklungsverlauf.

2. Commit-Identität

Dieser Punkt ist *entscheidend* für das Verständnis:

```
# Vor dem Rebase
git log --oneline feature
# abc1234 Y - Feature fertiggestellt
# def5678 X - Feature begonnen

# Nach dem Rebase
git log --oneline feature
# 111aaaa Y' - Feature fertiggestellt  ← NEUER HASH!
# 222bbbb X' - Feature begonnen        ← NEUER HASH!
```

Die Commits nach dem Rebase sind **komplett neue Git-Objekte**. Sie haben:

- denselben Autor
- dieselbe Commit-Message
- dieselben Code-Änderungen (Diffs)

- aber einen anderen SHA-Hash (weil der Parent-Commit anders ist)

3. Konfliktbehandlung

Bei einem Merge musst du Konflikte *einmal* lösen – im Merge-Commit. Die Konfliktlösung wird Teil dieses einen Commits.

Bei einem Rebase musst du Konflikte *für jeden Commit einzeln* lösen, während Git die Commits nacheinander auf die neue Basis anwendet. Das kann aufwändiger sein, führt aber dazu, dass jeder Commit für sich genommen „funktioniert".

```
# Rebase mit Konflikten
git rebase main
# CONFLICT in datei.php
# Konflikt lösen, dann:
git add datei.php
git rebase --continue
# Möglicherweise nächster Konflikt im nächsten Commit...
```

4. Auswirkungen auf andere Entwickler

Hier liegt der **kritischste Unterschied**:

Szenario	Merge	Rebase
Commits noch nicht gepusht	✅ Sicher	✅ Sicher
Commits bereits gepusht	✅ Sicher	⚠ **Gefährlich!**
Andere arbeiten auf dem Branch	✅ Sicher	❌ **Sehr problematisch!**

Wenn du Commits rebasst, die bereits gepusht wurden, änderst du deren SHA-Hashes. Wenn ein Kollege diese Commits bereits hat, entstehen *Duplikate* in der Historie, weil Git die alten und neuen Commits als unterschiedlich betrachtet.

Wann sollte ich Merge verwenden?

Merge ist die richtige Wahl, wenn:

1. **Du öffentliche Historie integrierst** – Wenn du Änderungen aus `main` in deinen Feature-Branch holst und dieser Branch bereits gepusht wurde, ist Merge sicherer.
2. **Mehrere Entwickler am gleichen Branch arbeiten** – Ein Merge verändert keine bestehenden Commits, sodass niemand Probleme bekommt.
3. **Du die komplette Entwicklungshistorie bewahren möchtest** – Für Auditing oder Nachvollziehbarkeit kann die „echte" Historie wichtig sein.
4. **Du größere Feature-Branches in `main` integrierst** – Viele Teams bevorzugen hier einen Merge-Commit als „Markierung", dass ein Feature abgeschlossen wurde.
5. **Du unsicher bist** – Merge ist die sicherere Option. Im Zweifel: Merge verwenden.

Beispiel: Feature in main mergen

```
git checkout main
git merge feature --no-ff  # --no-ff erzwingt einen Merge-Commit
```

Das `--no-ff` (no fast-forward) Flag ist wichtig, wenn du *explizit* einen Merge-Commit möchtest, auch wenn ein Fast-Forward möglich wäre.

Wann sollte ich Rebase verwenden?

Rebase ist die richtige Wahl, wenn:

1. **Du deine lokale Arbeit aufräumen möchtest, bevor du sie teilst** – Vor dem ersten Push ist Rebase völlig sicher und macht deine Commits präsentabler.
2. **Du deinen Feature-Branch auf den neuesten Stand von `main` bringen möchtest** (und der Branch noch nicht gepusht wurde oder nur du daran arbeitest).
3. **Du eine lineare, saubere Historie bevorzugst** – Manche Teams haben die Konvention, dass Feature-Branches vor dem Merge rebased werden müssen.
4. **Du mit Interactive Rebase Commits zusammenfassen oder aufteilen möchtest** – Das ist nur mit Rebase möglich.

Beispiel: Feature-Branch aktualisieren

```
git checkout feature
git fetch origin
git rebase origin/main
```

```
# Jetzt ist dein Feature-Branch auf dem neuesten Stand
```

In PhpStorm kannst du das über **Git → Rebase...** machen oder über das Git-Log-Fenster (Rechtsklick auf `origin/main` → „Rebase Current onto Selected").

Der kombinierte Workflow: Das Beste aus beiden Welten

Viele professionelle Teams nutzen einen **kombinierten Workflow**, der die Vorteile beider Ansätze vereint:

```
flowchart TD
    A["Feature-Branch erstellen"] --> B["Lokal entwickeln\nmit kleinen Commits"]
    B --> C{"Bereit zum Teilen?"}
    C -->|Nein| B
    C -->|Ja| D["Interactive Rebase:\nCommits aufräumen"]
    D --> E["Rebase auf aktuellen main"]
    E --> F["Push Feature-Branch"]
    F --> G["Pull Request erstellen"]
    G --> H["Code Review"]
    H --> I{"Änderungen nötig?"}
    I -->|Ja| J["Änderungen committen"]
    J --> H
    I -->|Nein| K["Merge oder Squash-Merge\nin main"]

    style D fill:#FFD700,color:#000000
    style E fill:#FFD700,color:#000000
    style K fill:#90EE90,color:#000000
```

Die Schritte im Detail

1. **Während der Entwicklung:** Mache so viele kleine Commits wie nötig, ohne dir über die „Schönheit" der Historie Gedanken zu machen.
2. **Vor dem Push:** Nutze Interactive Rebase (`git rebase -i`), um deine Commits aufzuräumen – Work-in-Progress-Commits zusammenfassen, Commit-Messages verbessern.

3. **Vor dem Pull Request:** Rebase auf den aktuellen `main`, um sicherzustellen, dass dein Branch sauber auf dem neuesten Stand aufbaut.
4. **Beim Mergen:** Der Pull Request wird entweder mit einem Merge-Commit oder einem Squash-Merge in `main` integriert.

Die „goldene Regel" des Rebasings

> **Rebase niemals Commits, die bereits gepusht wurden und an denen andere arbeiten könnten!**

Diese Regel ist so wichtig, dass sie einen eigenen Abschnitt verdient. Hier ist, was passiert, wenn du sie brichst:

Das Katastrophen-Szenario

```
sequenceDiagram
    participant Du
    participant GitHub
    participant Kollege

    Du->>GitHub: Push commits X, Y
    GitHub->>Kollege: Pull commits X, Y
    Note over Kollege: Kollege arbeitet<br/>basierend auf Y
    Du->>Du: Rebase X, Y → X', Y'
    Du->>GitHub: Force Push X', Y'
    Note over GitHub: X, Y ersetzt durch X', Y'
    Kollege->>GitHub: Push neuer Commit Z
    Note over GitHub: KONFLIKT!<br/>Z basiert auf Y,<br/>aber Y existiert nicht mehr
    GitHub-->>Kollege: Rejected!
    Note over Kollege: Muss jetzt manuell<br/>reparieren
```

Dein Kollege hat jetzt ein ernsthaftes Problem: Seine Arbeit basiert auf Commits, die in der offiziellen Historie nicht mehr existieren.

Was tun, wenn es doch passiert ist?

Falls du versehentlich gepushte Commits rebased hast:

1. **Kommuniziere sofort mit deinem Team** – Informiere alle, die betroffen sein könnten.
2. **Koordiniere die Reparatur** – Alle müssen ihre lokalen Branches aktualisieren:
   ```
   git fetch origin
   git reset --hard origin/feature
   # ACHTUNG: Lokale Änderungen gehen verloren!
   ```
3. **Lerne daraus** – Richte Protected Branches ein, die Force-Push verbieten.

Merge und Rebase in PhpStorm

Merge durchführen

1. Stelle sicher, dass du auf dem Ziel-Branch bist (z.B. `feature`)
2. Gehe zu **Git → Merge...**
3. Wähle den Branch, den du einmergen möchtest (z.B. `main`)
4. Klicke auf **Merge**
5. Bei Konflikten öffnet sich der Merge-Dialog

Rebase durchführen

1. Stelle sicher, dass du auf dem Branch bist, den du rebasen möchtest (z.B. `feature`)
2. Gehe zu **Git → Rebase...**
3. Wähle den Branch, auf den du rebasen möchtest (z.B. `main`)
4. Klicke auf **Rebase**
5. Bei Konflikten:
   - Löse jeden Konflikt im Editor
   - Klicke auf **Continue Rebase** in der Notification

Tipp: Rebase über das Git-Log-Fenster

Eine besonders intuitive Methode in PhpStorm:

1. Öffne das **Git**-Tool-Window (Alt+9 / Cmd+9)
2. Sieh dir das Commit-Log an
3. Rechtsklicke auf den Commit, auf den du rebasen möchtest
4. Wähle **Rebase Current onto Selected**

Praktisches Beispiel: Ein Tag im Leben eines Entwicklers

Lass mich einen typischen Workflow durchspielen:

Morgens: Feature-Branch erstellen

```
git checkout main
git pull
git checkout -b feature/user-profile
```

Während des Tages: Entwickeln mit vielen kleinen Commits

```
git commit -m "WIP: Grundstruktur angelegt"
git commit -m "WIP: Formular hinzugefügt"
git commit -m "Fix Typo"
git commit -m "WIP: Validierung"
git commit -m "Fertig mit Validierung"
```

Vor dem Feierabend: Aufräumen mit Interactive Rebase

```
git rebase -i HEAD~5
# Im Editor: squash die WIP-Commits zusammen
# Ergebnis: 2 saubere Commits statt 5 chaotische
```

Nächster Morgen: Auf aktuellen main rebasen

```
git fetch origin
git rebase origin/main
# Konflikte lösen, falls nötig
```

Push und Pull Request

```
git push -u origin feature/user-profile
# Pull Request auf GitHub erstellen
```

Nach dem Review: Merge in main

Auf GitHub: **Squash and Merge** oder **Merge Commit** – je nach Team-Konvention.

Zusammenfassung: Merge vs. Rebase auf einen Blick

Kriterium	Merge	Rebase
Historie	Verzweigt, zeigt parallele Entwicklung	Linear, als wäre sequentiell entwickelt worden
Neue Commits	Ein Merge-Commit	Keine neuen Commits (außer neu erstellten)
Original-Commits	Bleiben unverändert	Werden neu erstellt (neue SHA-Hashes)
Sicherheit	Immer sicher	Nur sicher für nicht-gepushte Commits
Konfliktlösung	Einmal im Merge-Commit	Für jeden Commit einzeln
Lesbarkeit	Kann unübersichtlich werden	Sehr übersichtlich
Empfohlen für	Öffentliche Branches, Team-Arbeit	Lokale Arbeit, Branch-Aktualisierung

Meine Empfehlung: Starte mit Merge als Standard – es ist sicherer und verzeihender. Nutze Rebase gezielt für lokale Aufräumarbeiten und Branch-Aktualisierungen, *bevor* du pushst. Mit der Zeit wirst du ein Gefühl dafür entwickeln, wann welcher Ansatz am besten passt. □□

Revision #1
Created 2026-06-29 15:38:24 UTC by art10m
Updated 2026-06-29 15:41:08 UTC by art10m